High Performance Implementations of the RSA Algorithm

Vishnu Kumar Orathi Pathangi, Pavan Kumar Tipparti

Department of Computer Science

St. Cloud State University
720 Fourth Avenue South,
St. Cloud, Minnesota 56301-4498

jherath@stcloudstate.edu

Abstract

In designing crypto systems the emphasis is not only on security, but also on speed. Cryptographic algorithms are more efficiently implemented in custom hardware than in software running on general-purpose processors. We have been experimenting with methods to improve undergraduate research experiences and the quality of teaching crypto systems in security and architecture courses for Computer Science and Information Systems students. Mapping algorithms to special purpose hardware is an important concept to learn in any Computer Security course. This paper describes the learning module developed to help students understand asymmetric key cryptographic algorithms and architectures. It will focus on the software-to-hardware transformation of asymmetric key cryptographic algorithms, recent advances in high speed hardware implementations of the RSA algorithm and our experiences incorporating such concepts in Computer Security and Architecture courses.

1. Introduction

In 1977, Scientific American published a $100 award for decrypting a 4 x 32 cypher text matrix, the first RSA challenge. The public key was 129 decimal digits long and e was 4 digits long. In 2002 Rivest, Shamir and Adelman received the ACM Turing Award for introducing the RSA algorithm and their contributions to cryptography.

In 1977 R.L Rivest, A. Shamir and L. Adleman (RSA) came up with a new technique for security similar to our day-to-day life paper documentation, i.e, our information is private and the transactions that we do are signed. These two methods, when done digitally, have a great advantage. The signatures cannot be shorn off or counterfeited, making this a perfect application for electronic banking and smart card applications. [7].

The RSA asymmetric key algorithm is based on modular exponentiation. This modular exponentiation is also related to the factorization of large integers. Today's RSA challenge is factorizing large numbers. In 1974, the largest number factorized was 45 decimal digits long. There is a $10,000 award for factorizing 174 digits long decimal numbers and $200,000 award for factoring 617 decimal digit long numbers [14].

Finding the exponent of large numbers is a difficult problem to solve. Similarly, factorizing large numbers is a difficult problem to solve. This paper addresses the problem of finding modular exponents efficiently.

Many applications depend on the security provided by the RSA algorithm. Web services provide a wide range of applications extending from e-mails and online banking to commercial business. Most of web-based authentication uses single factor authentication, either a password or a pin, which makes it easy to perpetrate account hijacking. The FDIC, Federal Deposit Insurance Corporation, suggests two-factor authentication methods, a password or pin along with an authenticator like a pass code, for preventing or minimizing identification theft [2]. Business organizations are looking for cost effective techniques to preventing identification theft.

Hardware-based security functions have proved to be more secure with higher performances. Such functions are not only resistant to code breakers, but are not restricted by a fixed block size, making them useful in database applications. Some hardware-based cryptosystems can be programmed to change the vendor code every 60 seconds and hence the pass code changes [8].

The RSA algorithm has proved to be one of the best methods for preventing ID theft, as it uses two-factor authentication methods for encryption and decryption. High-speed implementations of the RSA algorithm are necessary in applications such as smart cards and SIM cards in cell phones. The RSA algorithm, when used as a 'software only' implementation by an end user, proves to be weak when compared with hardware and software combined application implementation. When implementing the 'software only'

method, the application becomes static and the reused password becomes an easy target for crackers.
A hardware implementation of the RSA algorithm is more secure and faster when compared with a software implementation. The faster the cryptosystem, the safer the encryption is. The probability of the password being cracked in a hardware-based implementation is less. Many crypto systems consist of key sizes varying between 1024 bits to 2048 bits. This increasing key size tends to increase the length of the modulus and hence the computation time. RSA security primarily depends on modular exponentiation, performed by a series of modular multiplications. Hence one of the problems in hardware implementation is finding a high performance circuit for modular exponentiation.

This paper focuses on the software-to-hardware transformation of asymmetric key cryptographic algorithms. Section 2 describes the RSA algorithm with examples. Section 3 reviews left-to-right binary exponentiation algorithm and architectures. Section 4 presents right-to-left binary exponentiation algorithm and architectures. Section 5 presents Montgomery multiplication. These three sections present software-to-hardware transformation, sequential and parallel implementations. These examples can be easily used in a classroom to help students understand the implementation of modular exponentiation. Section 6 provides a summary.

2. The RSA Algorithm

Diffie and Hellman first introduced the concept of a public key in 1976 [10]. In this public key cryptosystem (PKC) the encryption and decryption are done with two different keys. RSA uses either of the two keys for encryption. One of the main advantages of PKC is that it provides confidentiality and key management. The characteristics of PKC not only depend on the keys, but also on the algorithm used.

The basic operation of the RSA algorithm is as follows [10]:

The message *M* to be sent is first represented in integer form (between 0 and n-1). In general, the representation of *M* should be of block size less than or equal to $\log_2(n)$.

The private key or the secret key consists of two large prime numbers *p* and *q* and two exponents *e* and *d*.

The cipher text $C = M^e \pmod{n}$.

The decrypted text $M = C^d \pmod{n}$.

It is understood for the above equations that the encryption keys (public keys) are e and n and decryption keys (private keys) are d and n. It is understood that both the communicating parties would know the value of n. The sender would know e, while the receiver will know d.

The procedure for establishing keys for encryption and decryption are as follows:

- Compute n, a very large number such that $n = p*q$ and is difficult to factor
- Compute Euler's totient function $\varphi(n)$ such that $\varphi(n) = (p-1)*(q-1)$ ---(1)
    - Check that d satisfies $\gcd(d, \varphi(n)) = 1$
- Compute e such that $e*d \equiv 1 \bmod (\varphi(n))$. ---(2)

In general [7] a PKC encryption and decryption have the following properties:

- Given $M^{ed} = M \bmod n$ for all $M < n$, it is possible to find e, d, and n.
- For all $M < n$, the value of M^e and C^d can be calculated easily.
- If e and n alone are given, it is impossible to find d.

Example of RSA Encryption

The following example would help to understand the RSA algorithm more clearly. Let us assume that the keys were generated with the two prime numbers p = 5 and q = 11 then:

Step 1 calculate n

- $n = p*q = 5 * 11 = 55$

Step 2 Calculate the Euler's totient function $\varphi(n)$

- $\varphi(n) = (p-1)*(q-1) = 4*10 = 40$
- e should be relatively prime and less than $\varphi(n)$ so we can choose e = 3

Step 3 Compute d from equation (2)

- $e*d \equiv 1 \bmod 40$; hence d = 27

Now that we know e, d, and n, we can form the private and the public keys. The private key consists of d and n i.e. (27, 55) and the public key consists of e and n i.e. (3, 55).

Once the public key and the private keys are known, we can encrypt the message using the public key as $C = M^3 \bmod 55$.

The decryption process involves the use of private keys and the encrypted message can be retrieved from $M = C^{27} \bmod 55$.

Assume that the plain text M is the integer 12.
Then to encrypt M we use $C = M^e \bmod n$, i.e,

$C = 12^3 \bmod 55$
$= [(12 \bmod 55) * (12^2 \bmod 55)] \bmod 55$
$12 \bmod 55 = 12$
$12^2 \bmod 55 = 144 \bmod 55 = 34$, therefore,
$C = [12 * 34] \bmod 55 =$ **23**

To decrypt the cipher text we use $M = C^d \bmod n$

$M = 23^{27} \bmod 55$
$= [(23^3 \bmod 55) * (23^8 \bmod 55) * (23^8 \bmod 55) * (23^8 \bmod 55)] \bmod 55$
$23^3 \bmod 55 = 12167 \bmod 55 = 12$
$23^8 \bmod 55 = [23^3 \bmod 55 * 23^3 \bmod 55 * 23^2 \bmod 55] \bmod 55$

$23^8 \bmod 55 = [12*12*34] \bmod 55 = 1$

⇨ $M = [12*1*1*1] \bmod 55 = \mathbf{12}$

Initially we selected M to be 12, and after encrypting and decrypting we have the initial value of M.

RSA key sizes currently vary between 512 bits to 2048 bits. This increasing key size tends to increase the length of the modulus and hence the computation time. It also affects modular exponentiation, register length and the size of the adders. The following sections describe various algorithms and architectures that are used in solving the modular exponentiation and multiplication problems.

3. LR Binary Exponentiation Algorithm and Architecture [3]

Consider the encryption of a plain text M to C, $C = M^e \bmod n$ where M multiplies by M 'e' times and then finally with the modulus function to get the cipher text C. There are several methods proposed, one of which is the Binary exponentiation method. In this method we convert the exponent from decimal to binary and thus use the Left to Right or Right to Left methods to perform the multiplication.

LR Binary Exponentiation

In the Left to Right binary exponentiation method, the 'e' is converted from decimal to binary b_i bits and arranged from MSB to LSB. Then the modular squaring is performed for each bit [3].

Example LR Binary Exponentiation

Let us calculate $C = 23^{25} \bmod 55$ where e =25; M =23; n = 55 then by LR binary exponentiation algorithm

$e = (25)_{10} = (1 \quad \mathbf{1} \quad \mathbf{0} \quad \mathbf{0} \quad 1)_2$

e4 **e3** e2 e1 e0

$C = ((((23 \bmod 55)^2 * 23 \bmod 55)^2 \mathbf{\bmod 55})^2 \mathbf{\bmod 55})^2\, 23 \bmod 55$

$C = 23^{25} \bmod 55 = 12$

Different colors represent the relationship in steps of the computation.

Algorithm

The algorithm for LR Binary exponentiation is presented below:

For the given input message *M*, exponent *e*, modulus *n* the cipher text *C* is represented as $C = M^e \pmod n$

1. If $e_{bi-1} = 1$ then C = M else C = 1
2. For i = bi – 2 down to 0

   2a. $C = C*C \pmod n$

   2b. If $e_i = 1$ then $C = C*M \pmod n$
3. Return C

The complete computation of the exponentiation is shown in Table1.

i	e_i	Step 2a	Step 2b
3	1	23^2	$23^2*23 = 23^3$
2	0	23^6	23^6
1	0	23^{12}	23^{12}
0	1	23^{24}	$23^{24} *23 = 23^{25}$

Table 1: Computation for LR Binary exponentiation method

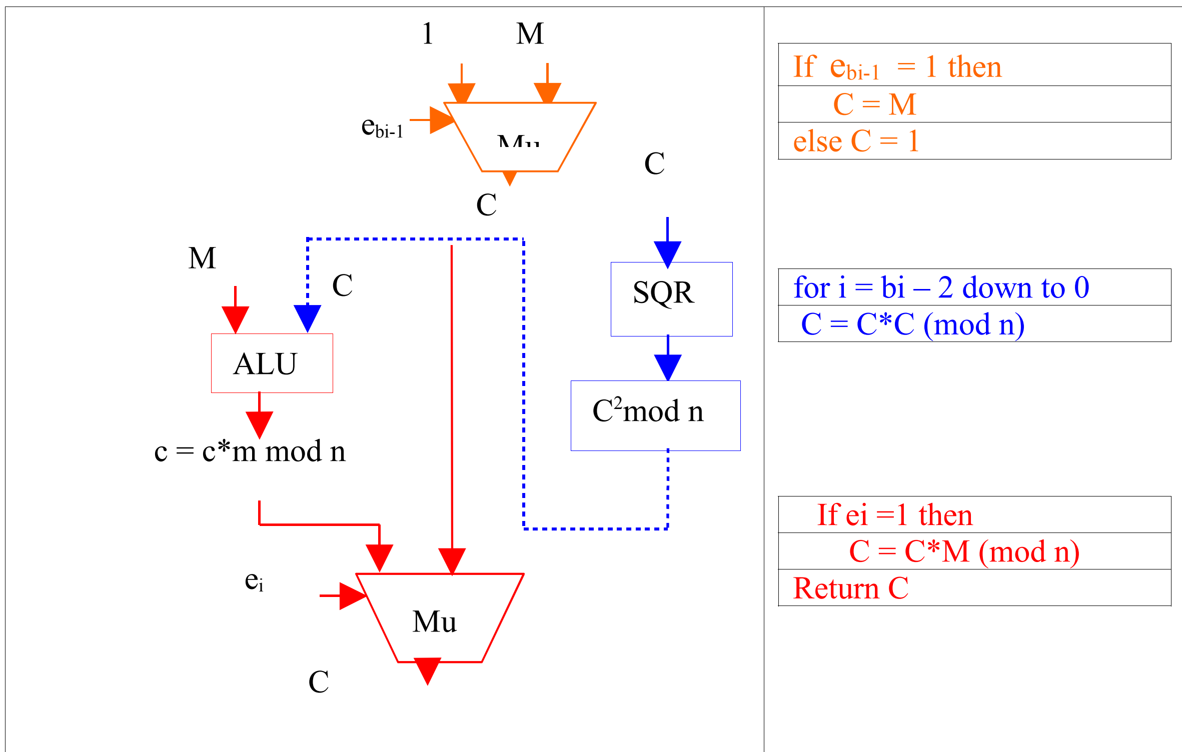

Figure 1: Hardware Implementation of the LR Binary Exponentiation Method

The hardware architecture of LR exponentiation is presented in Figure 1. This method requires two registers, one to store M and other to store the value of C. Different colors are used in Figure 1 to help understand the software-to-hardware transformation of If

Then .. Else ..and For loop expressions. Many software-to-hardware transformations for symmetric key algorithms are presented in [16].

4. RL Binary Exponentiation Algorithm and Architecture [3]

The Right to Left Binary Exponentiation is similar to Left to Right Binary Exponentiation, except that the computations are performed with e from LSB to MSB.

Example RL Binary Exponentiation

Let us consider the same example used for LR Binary Exponentiation to show the working of the RL Binary Exponentiation method:

e =25; M=23; n= 55

$e = (25)_{10} = (\ \mathbf{1}\ \ \mathbf{1}\ \ \mathbf{0}\ \ \mathbf{0}\ \ \mathbf{1}\ )_2$

e4 e3 e2 e1 e0

((**23**) **1** **1** (**23^8 mod 55**)(**23^{16}mod55**))

$C = 23^{25} \text{mod } 55 = 12$

Different colors represent the relationship in steps of the computation.

RL Binary Exponentiation Algorithm

Let us assume the input message to be *M*, *e* as the exponent, and the modulus *n* then the output of the encrypted message is $C = M^e$ mod n. In the RL binary method, we initially make the value of C to be 1 and another register R to hold the value of the powers of M every time it is computed.

1. C =1; R = M
2. For i = 0 to bi – 2
   2a. If e_i = 1 then C = C*R mod n
   2b. R = R*R mod (n)
3. If e_{bi-1} = 1 then C = C*R mod n
4. Return C

Calculations of the above problem are shown in Table 2.

i	bi	Step 2a	Step 2b
0	1	1*23 = 23	23^2
1	0	23	23^4
2	0	23	23^8
3	1	23^8*23	23^{16}
4	1	**$23^9 * 23^{16} = 23^{25}$**	

Table 2: Calculations for the RL Binary Exponentiation Method

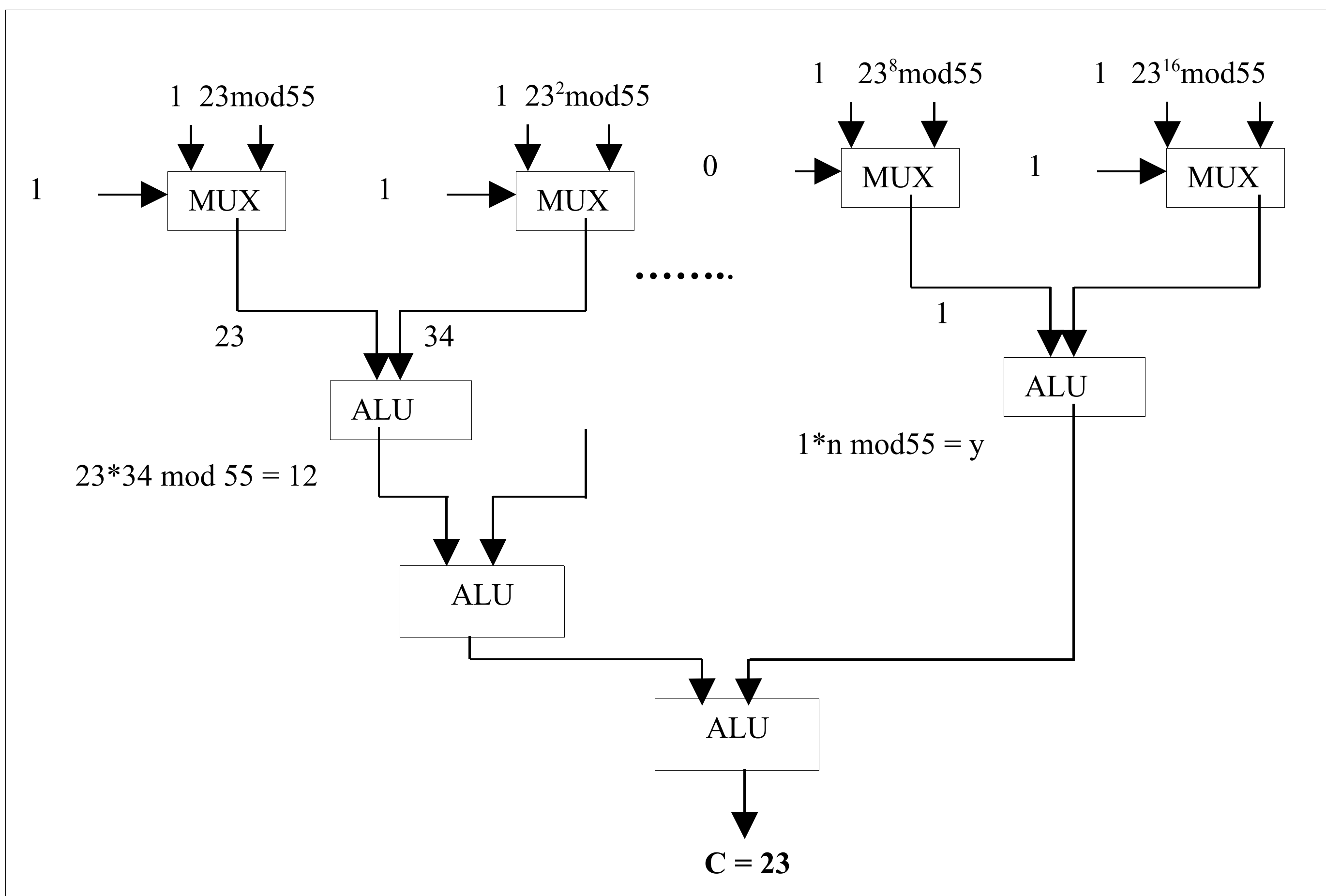

Figure 2: Parallel Hardware Implementation of RL Exponentiation

The function-to-hardware transformation provides simple and direct mapping for parallel high performance implementation. Figure 2 represents the parallel hardware for RL Binary Exponentiation. Figure 3 shows the general hardware for RL Binary Exponentiation. This computation can be represented as an iteration consisting of If Then

Else and For loop expressions. This sequential implementation reduces space, cost and performance.

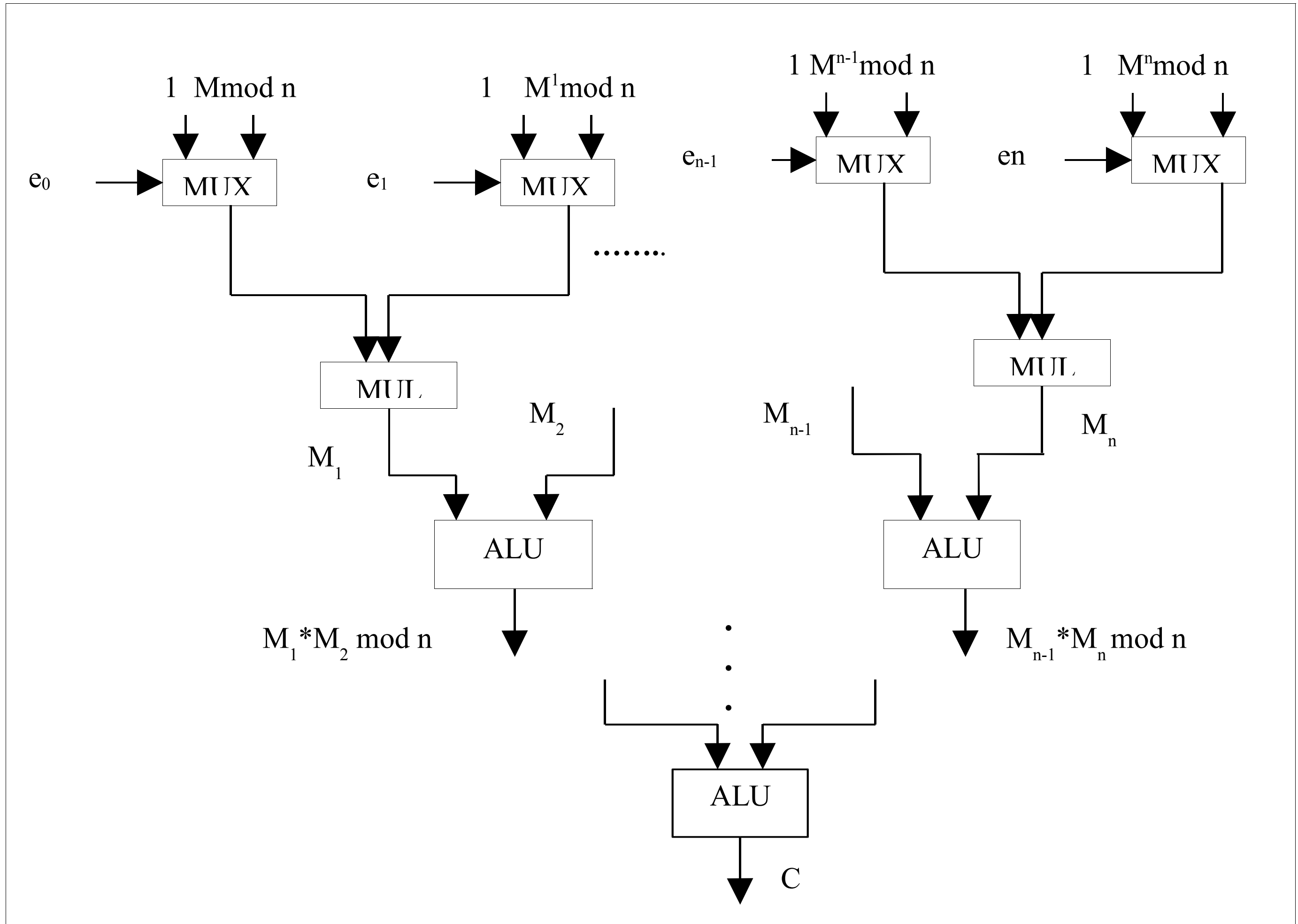

Figure 3: Hardware Implementation of RL Binary Exponentiation Method

5. Montgomery Multiplication Algorithm

This algorithm was introduced by Peter L. Montgomery in 1985 [15]. This method was used primarily for modular arithmetic reductions. The Montgomery method is used to find the modulo of a large number multiplied by the exponentiation of two numbers similar to $M = a^b \pmod n$.

The Classical method involved computing the product of the integers and then reducing the result modulo n, whereas the Montgomery method involves multiplication and shifting, rather than division, making it more efficient than the Classical method.

Formally, if n, T, R are three positive integers such that

- R > n, gcd (n, R) = 1; n and R should be relatively prime (hence should be odd)
- n * R >T >= 0; Then T mod n with respect to R = $TR^{-1} \pmod n$.

Hardware implementation of the Montgomery Algorithm is discussed in [3].

Algorithm
Let A and B be two positive integers, then the Montgomery product is $A*B*r^{-1} \bmod n$. Montgomery proposed MonPro ( ) function to describe the algorithm.

The description of the algorithm to compute MonPro (A, B) using Binary add-shift algorithm is as follows:

- $r = 2^K$ A, B < K
- $A = (A_{k-1}, A_{k-2} \ldots A_0)$
- MonPro (A, B) = $2^{-k}. (A_{k-1}, A_{k-2} \ldots A_0).B$

$$\Rightarrow 2^{-k} \sum_{i=0}^{k-1} A_i.2^i.B (\bmod n)$$

- Let $t = (A_0 + 2A_1 + \ldots + 2^{k-1}A_{k-1}).B$
- Let u = t *mod n

Computation of‘t’:

1. t = 0
2. For i = k-1 to 0
   2a. t = t + Ai * B
   2b. t = 2*t

When in summation 2-k direction is reversed
1. t = 0

2. For i = 0 to k -1
   2a. $t = t + A_i * B$
   2b. t = t/2

Computation of 'u':

$u = t * \bmod n = 2^{-k} * A * B \pmod n$

This can be done by either subtracting *n* during every add-shift step or by changing *u* to be even. i.e., u = u + n

$u = u*2^{-1} \pmod n$

The Binary shift add algorithm is used to compute *u* is as follows:

1. u = 0
2. For i = 0 to k -1
   2a. u = u + Ai*B
   2b. if u is odd
       u = u + n;
3. u = u/2

- Combine 2a and 2b to compute the least significant bit, u_0 of u. $u_0 = u_0$ xor (Ai*B)

The Montgomery Algorithm function from [3] is given below in Table 3.

MonExp (M, e, n)	**MonPro(a, b)**

1. Compute n' (Extended Euclidean method)
2. M' = M *r*mod n
3. x' = 1* r* mod n
4. for i = k-1 down to 0
    1. x' = MonPro (x',x')
   4a. if ei = 1
     x' = MonPro (M',x')
5. x = MonPro (x',1)
6. return x
	1. t = (a'*b')
	2. m = t *n' mod r
	3. u = (t +m *n) / r
	4. if u >= n
	    1. Then u = u – n
	5. else u = u

Table 3: Montgomery Algorithm and Function

Example

Consider computing $x = 23^{25} \bmod 55$ using the above function

$x = 23^{25} \bmod 55$

n = 55; **r = 64** = $2^6 > n$

1. Compute n

(64 * 49) – (55 *57) = 1

r' = 49; **n' = 57;**

2. Compute M'

M' = M * r * mod n
   = 23 * 64 * mod 55

M’ = 42
3. x’= 1*r*mod n
=1*64*mod 55
x’= 9
$25_{10} = (11001)_2$

The calculations for step 5 and step 6 of the algorithms are shown in Table [3]

E	**Step 4**	**Step 4a**
1	MonPro(9,9)	MonPro(42,9)
1	MonPro(42,42)	MonPro(42,31)
0	MonPro(53,53)	
0	MonPro(31,31)	
1	MonPro(64,64)	MonPro(42,64)
		MonPro(42,1)

Table 3: Calculation Table for the Montgomery Algorithm

The expanded calculation of Table 3 for the *MonPro* function is given in Table 4.

MonPro(9,9) t = (x’ * x’) = (9*9) = 81 m = t * n’ mod r = 81 * 57 mod 64 = 9 u = (t + m * n)/r = (81 + 9*55) /64 = 9	MonPro(42,9) t = 42*9 = 378 m = 378*57 mod 64 = 42 u = (378 + 42*55)/64 = 42
MonPro(42,42) t = 1764 m = 1764*57 mod 64 = 4 u = (1764+4 *55)/64 = 31	MonPro(42,31) t = 1302 m = 1302*57 mod 64 = 38 u = (1302+ 38*55)/64 = 53
MonPro(53, 53) t = 2809 m = 2809* 57 mod 64 = 49 u = (2809 + 49*55) /64 = 86 u>n => 86-55 = 31	
MonPro(31,31) t = 961 m = 961* 57 mod 64 = 57 u = (961+57*55)/64 = 64	
MonPro(64,64) t = 4096 m = 4096*57 mod 64 = 0 u = (4096+ 0*55)/64 = 64	MonPro(42,64) t = 42*64 = 2688 m = 2688*57 mod 64 = 0 u = (2688 +0*55)/64 = 42

	MonPro(42,1) t = 42 m = 42*57 mod 64 = 26 u = (42+55*26)/64 = **23**

Table 4: Calculation Procedure for MonPro Function

6. Summary

This paper focused on helping students understand algorithms and architectures for the RSA algorithm. The RSA algorithm, numerical examples, RL Binary Exponentiation, LR Binary Exponentiation, iterative computations, function-to-hardware mapping and sequential and parallel architectures were presented. Also, we have discussed the speed, space and cost advantages of architectures. The Loop unrolled, parallel RL Binary Exponentiation architecture speeds up in RSA implementation.

Numerous papers discuss space and time tradeoffs. For example, by adding an extra RAM [6] implemented 1024-bit RSA exponentiation on a 32-bit processor core with execution time less than a second. The signed sliding window algorithm performs the exponentiation and multiplication. Different versions of Montgomery Multiplication algorithm were presented and compared space and time requirements in [1]. Algorithms analyzed in [1] include Separated Operand Scanning, Coarsely Integrated Operand Scanning, Finely Integrated Operand Scanning, Finely Integrated Product Scanning and Coarsely Integrated Hybrid Scanning. It appears that Coarsely Integrated Operand Scanning is better on general-purpose machines than the other algorithms discussed.

We have been experimenting with methods to improve undergraduate research experiences and the quality of teaching crypto systems in Security and Architecture courses for Computer Science and Information Systems students. The goal has been and continues to be to help them become good information assurance and security experts in a relatively short period of time, with both a theoretical understanding and practical skills, so that they can enter in and make valuable contributions to the profession. A learning module that can be used in the classroom based on the algorithms and architectures presented in this paper is at [12]. The Turing Award lecture [13] given by Adelman, Rivest and Shamir is a good starting point to present research related to RSA.

Acknowledgments

We sincerely thank Dr. Don Hamnes, Dr. S. Herath and Dr. J. Herath for their guidance, time, effort and feedback, in the writing of this paper.

References

[1] Cetin Kaya Koc, Toga Acar, Burton S. Kaliski Jr. “Analyzing and Comparing Montgomery Multiplication Algorithms”, IEEE Micro, 16(3):26 -33, June 1996.

[2] FDIC Technology Supervision Branch “Putting an End to Account-High jacking Identity Theft”, www.fdic.gov, December 14, 2004

[3] C.K. Koc, “High-speed RSA implementation”, Technical report TR201, RSA Laboratories, November 1994.

[4] A. Mazzeo, L.Romano, G.P.Saggese “FPGA-based Implementation of a serial RSA processor”, Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, pp: 10582, 2003

[5] David Naccache, David M’Ralhi, “Cryptographic smart cards”, IEEE Micro, June 1996.

[6] B.J.Phillips, N. Burgess, “Implementing 1024-bit RSA Exponentiation on a 32-bit Processor core”, asap, pp: 127, 12th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'00), 2000

[7] R.L Rivest, A.Shamir, L.Adleman “A Method for Obtaining Digital Signatures and Public Key Cryptosystems”, Communications of the ACM, Vol.21, Nr.2, S.120-126 1978.

[8] RSA Security Inc, http://www.rsasecurity.com/node.asp?id=1158.

[9] RSA Laboratories, http://www.rsasecurity.com/rsalabs/node.asp?id=2218

[10] William Stallings, “Cryptography and Network Security: Principles and Practice”, 3rd edition, Prentice Hall, 2003.

[11] B. Schneider, Applied Cryptography, New York: Wiley, 1996.

[12] http://web.stcloudstate.edu/jherath/csci697/RSAAcLearn.htm

[13] http://www.acm.org/awards/turing_multimedia/tl_mm_2002.htm

[14] http://www.rsasecurity.com/rsalabs/node.asp?id=2093

[15] P. L. Montgomery, “Modular Multiplication without Trial Division”, Mathematics of Computation, April 1985.

[16] Oi-Shong Chok “Computer Security Learning Laboratory: Implementation of DES and AES Algorithms using Spreadsheets”, MICS 2004